Chapter 2

Processes and Threads

While the specifics of how multiprocess, multithreaded programs execute differ greatly, all share some general characteristics. This chapter defines how TotalView looks as processes and threads. It also describes the way in which you tell the CLI to which processes and threads it should direct a command.

A Couple of Processes

When programmers write single-threaded, single process programs, they can almost always answer the question "Do you know where your program is?" These kind of programs are rather simple, looking something like this:

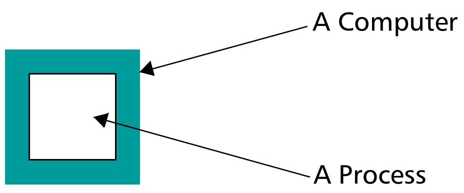

FIGURE 1: **A Uniprocessor**

If you use a debugger, any debugger, on something like this, you can almost always figure out what's going on. Before the program begins executing, you set a breakpoint, let the program run until it hits the breakpoint, then inspect variables to see what they've been set to. If you suspect there's a logic problem, you can step the program through its statements, seeing what happens and where things go and where things are going wrong.

A typical computing environment is actually a more complicated than this as the computer is actually existing a great number of programs. For example, your computing environment could have daemons and other support programs executing.

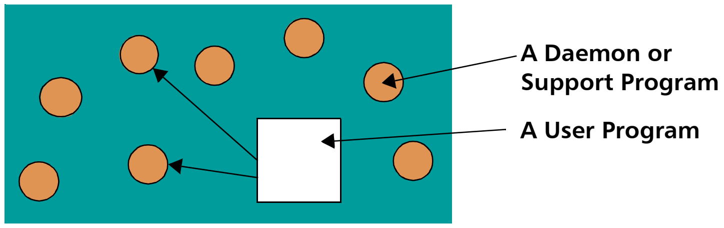

FIGURE 2: **A Program and Daemons**

These additional processes simplify a programmer's life because the application program no longer had to do everything itself. It could hand some things off to something else and they'd do the program's bidding.

This figure assumes that the application program only sends requests to the daemon. More complicated architectures are quite common. For example, the following figure shows an E-mail program communicating with a daemon on its computer. After receiving a request, this daemon sends stuff to an E-mail daemon on another computer which then delivers the stuff.

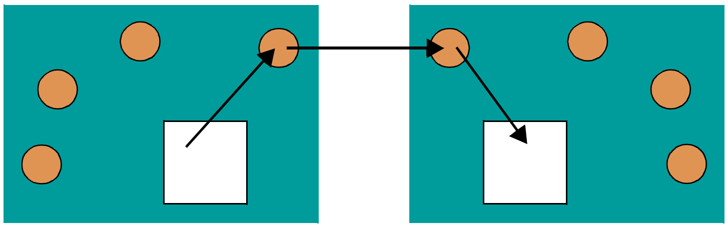

FIGURE 3: **Mail with Daemons**

This kind of processing assumes that the jobs being performed are disconnected. That is, no real cooperation exists between the processes. In all cases, one program hands work off to another. After the handoff occurs, there is no more interaction. While this is an extremely useful model, a more general model is that programs can divide up their work, parcelling it out to other computers. When this occurs, one program relies on another program to do some of its work. To gain any advantage, however, the work being sent to the second computer has to be work that the first computer doesn't need right away. In this way, the two computers could act more or less independently. And, because the first computer didn't have to do the work that the second computer did, the program could complete faster. (See Figure 4.)

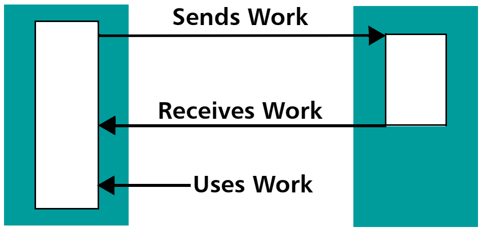

FIGURE 4: **Two Computers Working on One Problem**

Here's one the problems: because programs have bugs, how does a programmer debug what's happening on the second computer. A horrid solution is have a debugger running on each computer. A slightly better solution is to create a program the same way as was done with one computer, get it working, then split it up so it could use more than one computer. If done this way, there's a likelihood that any problems that occur will occur in the code that splits up the problem.

The TotalView solution is even better. It places a server on each processor as a program is launched. This server then communicates the "main" TotalView. This gives you one central location from which you can manage and examine all aspects of your program.

A Couple of Threads

The support programs just discussed are owned by the computer (actually, they're owned by the operating system). They do all sorts of activities from managing computer resources to providing services such as printing. If the operating system can have all sorts of things doing work for it, why can't a program? We'll call these new things *threads*. (See Figure 5.) This figure also shows, for the last time, the daemon processes that are executing. From now on, we'll just assume that they're are there.

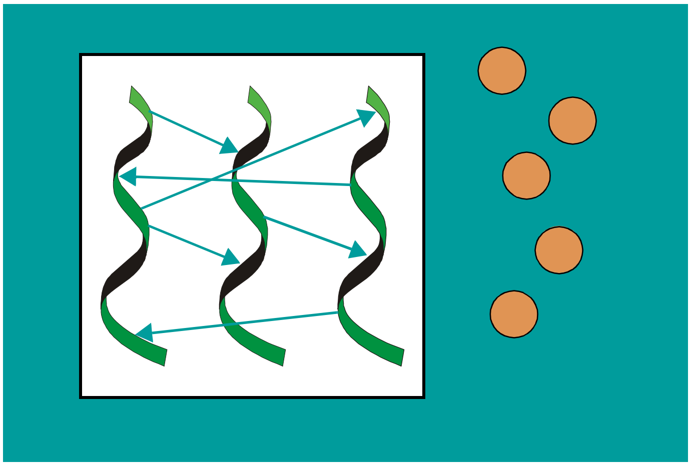
FIGURE 5: **Threads**

In this computing model, a program (the main thread) creates threads and these threads can also create threads if they need to. Each thread is a relatively independent process.

The debugging problem here is similar to the problem of processes running on different machines. In both cases, a debugger has to intervene with anything that is executing.

In the examples used so far, each executing process is doing something different and, except for when you need one thread to wait for another, its real difficult to tell what any thread is doing. And, the nature of these kinds of programs really keeps you away from having one "thread of execution"; that is, having only one thing running at a time.

Even More Complicated Programming Models

In the same way that software computing architectures become more complicated, advances in hardware design allowed the placement of more than one processor within a computer. So, expanding on the previous figure, you could be writing programs that execute in environments like what is shown in Figure 6.

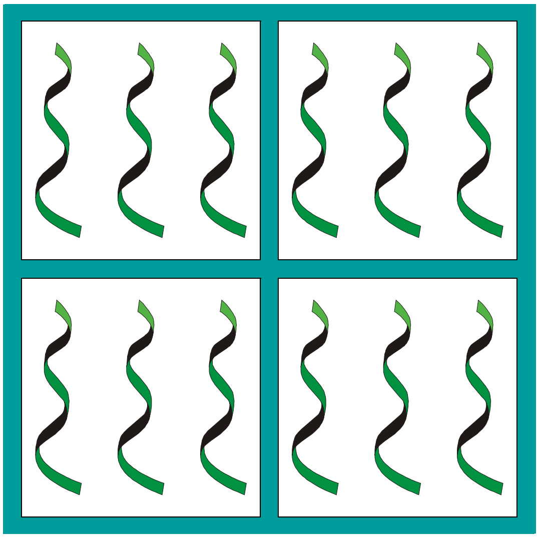

FIGURE 6: **Four Processor Computer**

This figure shows four linked processors on one board, each of which has three threads. This architecture could, in one sense, be thought of as an extension to the model of having more than one separate computer. And, if you think of your architecture like this, there's no reason that you can't join lots of computers together to solve problems. (See Figure 7.)

This drawing shows five computers, and each has four processors. Every program running on every processor has three threads, which means that altogether there are 60 user threads.

This drawing is just depicts processors and threads. It doesn't have any information about the nature of the programs and threads or even if the programs are the same or are different.

At any time, it is next to impossible to guess which threads are executing and what a thread is actually executing. And worse, protocols such as OpenMP are actually distributing and controlling the work being performed.

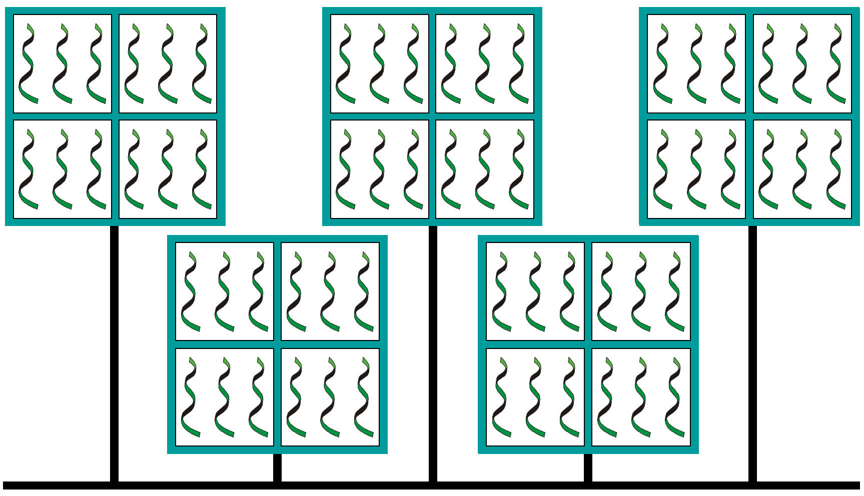

FIGURE 7: **Four Processor Computer Networks**

In these kinds of environments, a program (or the program within a library) is using another program to control how it distributes work across processors.

If everything goes right, things are real easy. When there's problems—and there are always problems—traditional debuggers and solutions are helpless. As you'll see, TotalView organizes this mass of executing procedures for you and, because operating systems can complicate things greatly, TotalView lets you correct problems in the ways operating systems misidentify or fail to identify threads.

More On Threads

All threads aren't the same. In Figure 8 (which is something like a figure shown earlier), the squiggly lines are user threads.

The threads that are part of your program are "user threads".

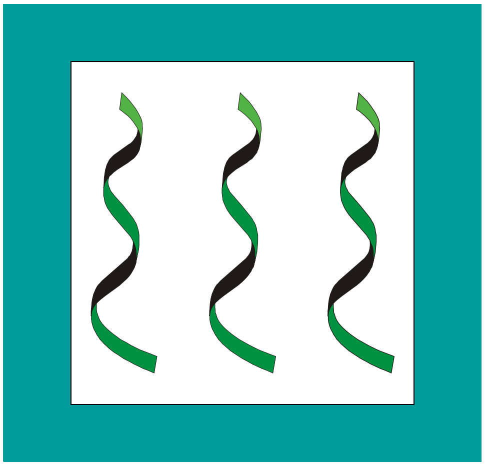
FIGURE 8: **Threads and Daemons**

NOTE Many computer architectures have something called "user mode", "user space", or something similar. "User threads" means something else. Without trying to be rigorous, a "user thread" is simply a process created by a program that does work for the program.

The threads that are part of the operating environment are "*manager threads*". Things would be nice and easy if this was all there was to it. Unfortunately, all threads aren't created equal and all threads don't execute equally. In most cases, a program creates manager-like threads. In the following figure, the threads that are lying down in their own room are user-created manager threads.

As these user-created manager threads are designed to perform a service for other threads in the program, they can be called "*service threads*".

The reason you need to know which of your user threads are actually service threads is that these kinds of threads perform different kinds of activities than your other user threads. Because their activities are so different, they are usually developed separately and aren't involved with the fundamental problems being solved by the program. For example, a service thread that dispatches messages sent from other threads may have bugs, but the bugs are of a different kind and the problems they have can most often been

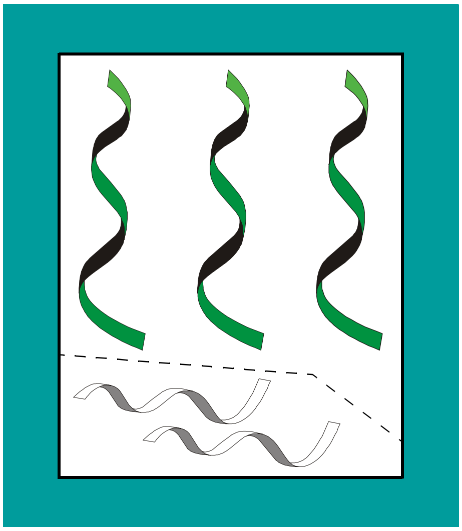

FIGURE 9: **User Threads, *Service* Threads, and Some Daemons**

dealt with separately than bugs that would occur in non-service user threads.

In contrast, your user threads are the agents performing your program's work and the interactions between them are where the action is. Being able to distinguish between the two kinds of threads means that you can focus on the threads and processes that are actively participating, rather than those sitting back, performing more subordinate activities.

Subdividing User Threads

Sometimes, TotalView gets lucky and it can identify which threads are performing service activities. In other cases, its not so lucky. While some threads can be identified (and on some architectures this may not be possible), you are often forced to identify which user threads are performing service activities.

In the following diagram, one of the three non-manager worker threads is event-driven.

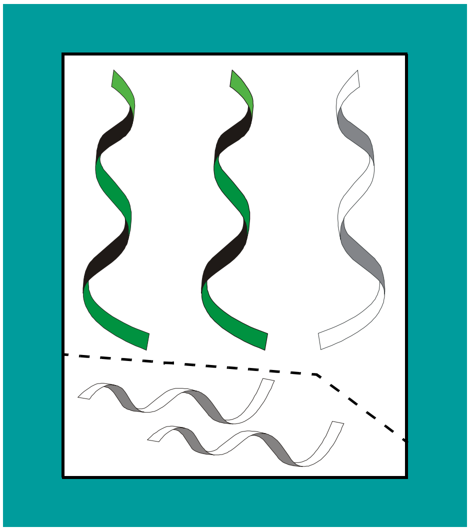

FIGURE 10: **Two Kinds of User Threads and Service Threads**

So, while this figure shows five threads, most of our efforts are going to be involved in debugging just two of them.

Organizing Chaos

While it is possible to attack the kinds of programs that are running thousands or processes across hundreds of computers one-at-a-time, it is not very practical. What TotalView does is organize these processes for you and then let you reorganize this information. The technical term for the way TotalView groups information is "group". Here are quick definitions of the four kinds of groups:

- **Control Group**: All the processes created by a program running on all processors. If your program uses processes that it didn't create, these other processes are in another program group.

- **Share Group**: All the processes within a control group that share the same code. In most cases, your program will have more than one share group. Share groups, like control groups, can have processes that execute on more than one processor.
- **Workers Group**: All the worker threads within a control group. These threads can be drawn from more than one share group.
- **Lockstep Group**: All threads that are at the same PC. This group is a subset of a workers group. Because all threads execute asynchronously, a lockstep group only exists for stopped threads. All threads in the lockstep group are also in a workers group.

In the list, the first two groups contain processes and last two groups contain threads. And, notice that "same code" means that the processes have the same executable file name.

TotalView's commands let you manipulate processes individually and by groups. In addition, you can create your own groups and manipulate a group's contents (to some extent).

NOTE Not all operating systems let you individually run a thread.

The following figure shows a processor running five processes (ignoring daemons and other programs not related to your program) and the threads within it. The figure indicates a control group and two share groups.

The elements in this figure are as follows:

Green (Gray in black and white) Area
: A CPU. Everything represented by this drawing exists within one processor.

White Rectangle
: Processes being executed by the CPU.

Control Group
: The five processes make up the control group. This diagram doesn't indicate which process is the **main** procedure.

Share Groups
: The control group has two share groups. The three processes in the first share group have the same executable. The two processes in the second share group also share a second executable.

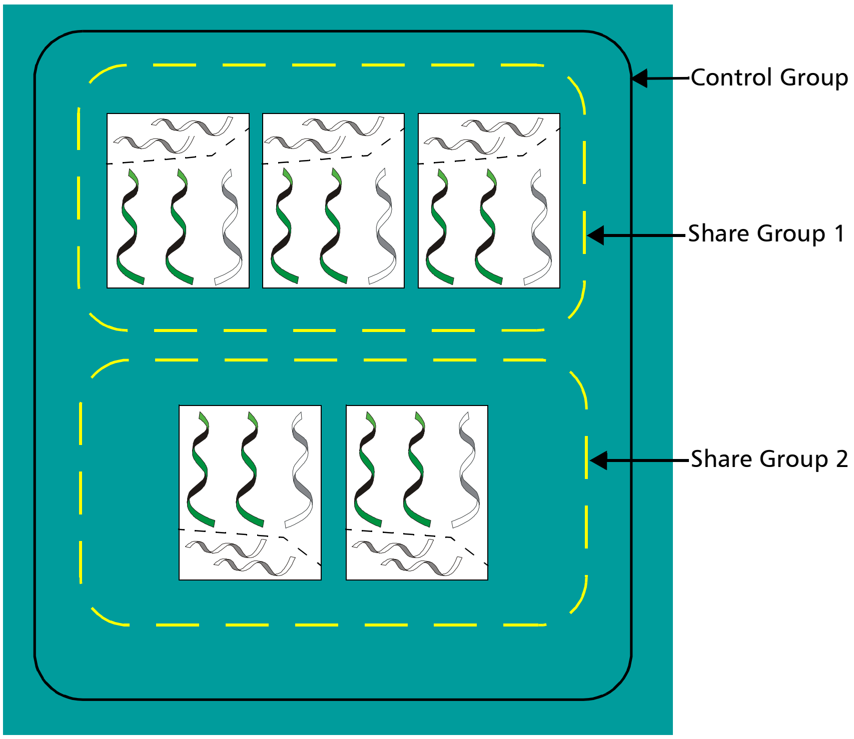

FIGURE 11: **Five Processors and Processor Groups**

Figure 12 looks at how the threads in this drawing are organized. As you can see, this figure adds the workers group and two lockstep groups. (You get extra credit if you know how many other lockstep groups there are.)

NOTE The control group is not shown as it encompasses everything in this figure.

Here is a description of the added elements in this figure:

Workers Group	All non-manager threads within the control group make up the workers group. Notice that this group includes service threads.
Lockstep Group	Each share group has its own lockstep groups. Graphically, two lockstep groups are indicated, one in each share group.
	If other threads are stopped, this picture indicates that they are not participating in either of these two lock-

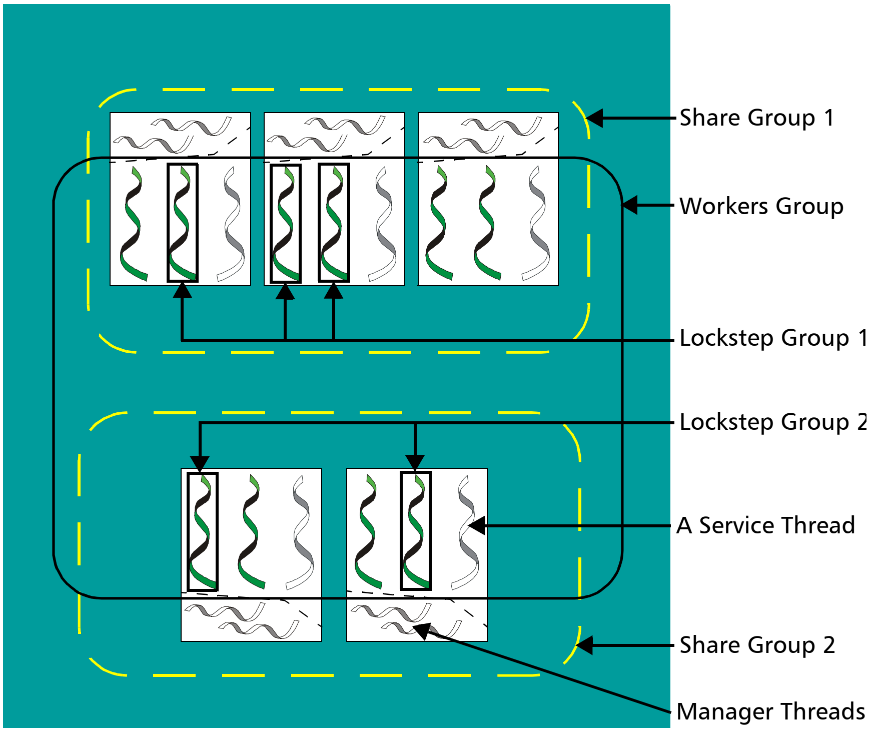

FIGURE 12: **Five Processors and Processor Groups**

step groups. Recall that a stopped thread is always in a lockstep group. (Its OK if a lockstep group has only one member.)

Service Threads Each process has one service thread. A process can have any number of manager threads. This figure, however, only shows one.

Manager Threads

The only threads that are not participating in the workers group are the ten manager threads.

Figure 13 extends the previous figure to show the same kinds of information executing on two processors.

This figure differs from the one its based on in that it has ten processes on two processors rather than five processes on one processor. However, the number of control groups and share groups is unchanged. This is not to say

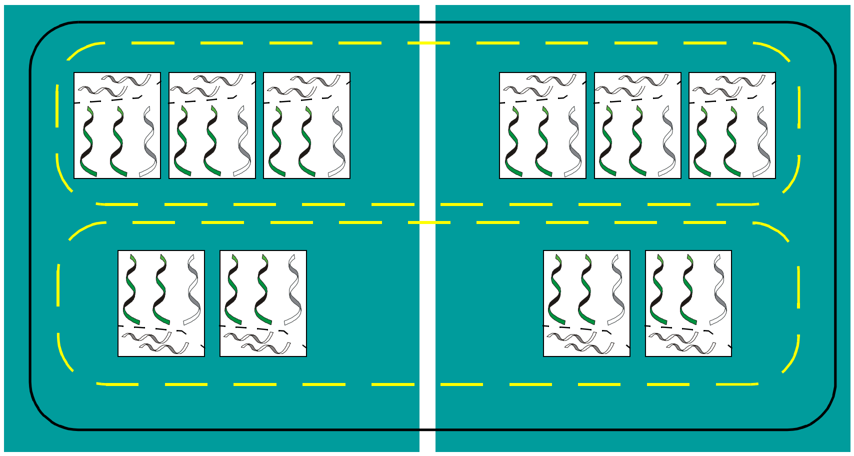

FIGURE 13: **Five Processes and Their Groups on Two Computers**

that it couldn't be different and that the processes on the second processor could not be part of a second control group. It's just in this example they aren't

Creating Sets

TotalView automatically creates and places items in groups as they are created. The exception is the lockstep groups which are created or changed whenever a program hits an action point. While there are many ways that this kind of organization can be built up, the following steps indicate the beginnings of how this might occur:

1 TotalView and your program are launched and your program begins executing within TotalView.

Control group: A group is created as the program is loaded.

Share group: A group is created as the program begins executing.

Workers group: The thread in the main routine is the workers group.

Lockstep group: There is no lockstep group.

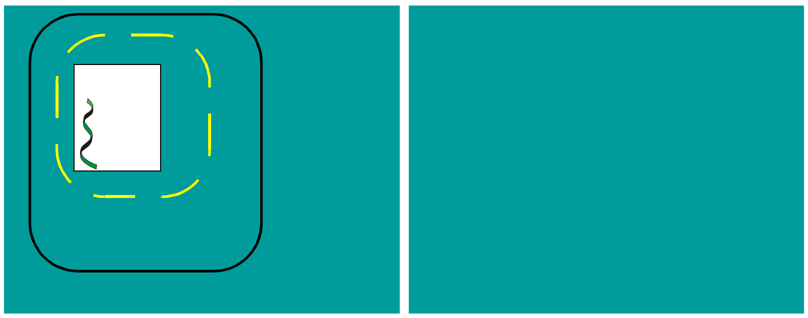

FIGURE 14: **Step 1**

2 The program forks a process.

Control group: A second process is added to the existing group.

Share group: A second process is added to the existing group.

Workers group: TotalView adds the thread in the second process to the existing group.

Lockstep group: There is no lockstep group.

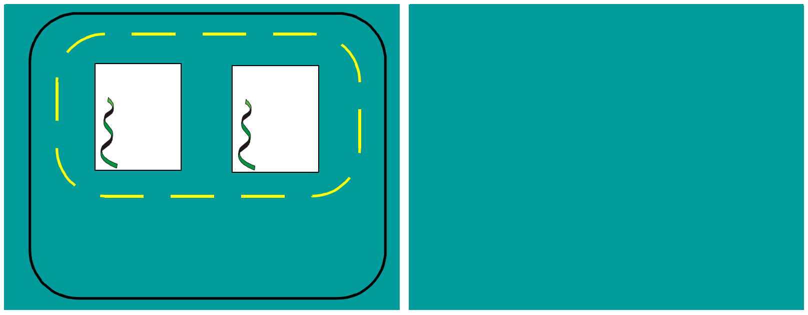

FIGURE 15: **Step 2**

3 The second process is exec'd.

Control group: The group is unchanged.

Share group: A second share group is created having this exec'd process as a member. This process is removed from the first share group.

Workers group: Both threads are in the worker's group.

Lockstep group: There is no lockstep group.

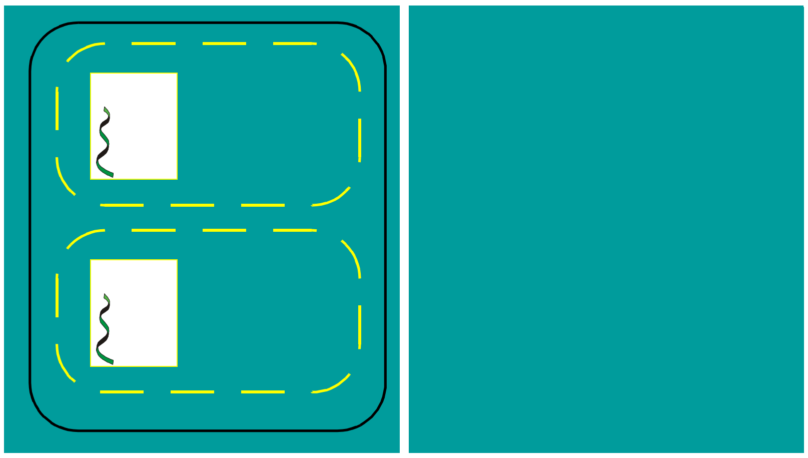

FIGURE 16: **Step 3**

4 The first process hits a break point.

Control group: The group is unchanged.

Share group: The groups are unchanged.

Workers group: The group is unchanged.

Lockstep group: A lockstep group is created whose member is the thread of the current process. (Each thread is its own lockstep group.)

5 A second version of your program is started from the shell. You attach to it within TotalView.

Control group: A third process is added.

Share group: This third process is added to the first share group.

Workers group: The thread in this third process is added to the group.

Lockstep group: There are no lockstep groups.

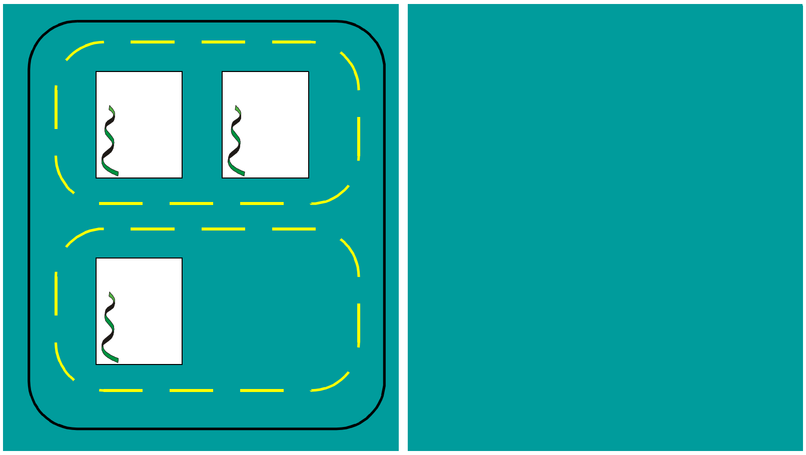

FIGURE 17: **Step 5**

6 Your program creates a process on another computer.

Control group: The control group is extended so that it contains this fourth process that is running on the second computer.

Share group: The first share group is extended to contain this fourth process that is running on the second computer.

Workers group: The thread within this fourth process is added to the workers group.

Lockstep group: There is no lockstep group.

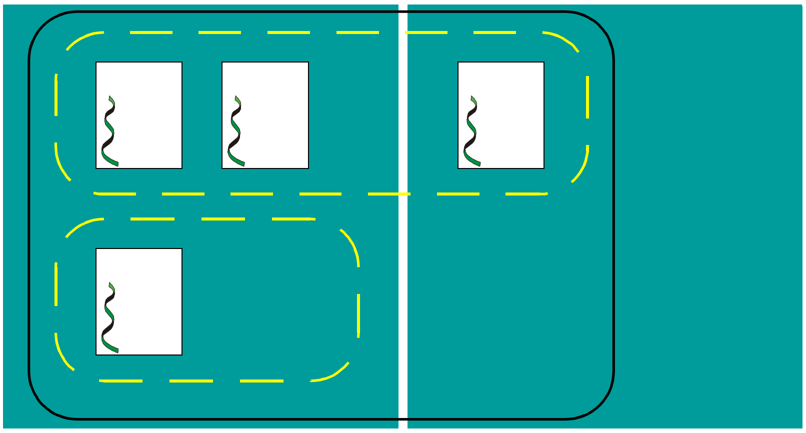
FIGURE 18: **Step 6**

7 A process within program group 1 creates a thread. This adds a second thread to one of the processes.

Control group: The group is unchanged.

Share group: The group is unchanged.

Workers group: A fourth thread is added to this group.

Lockstep group: There is no lockstep group.

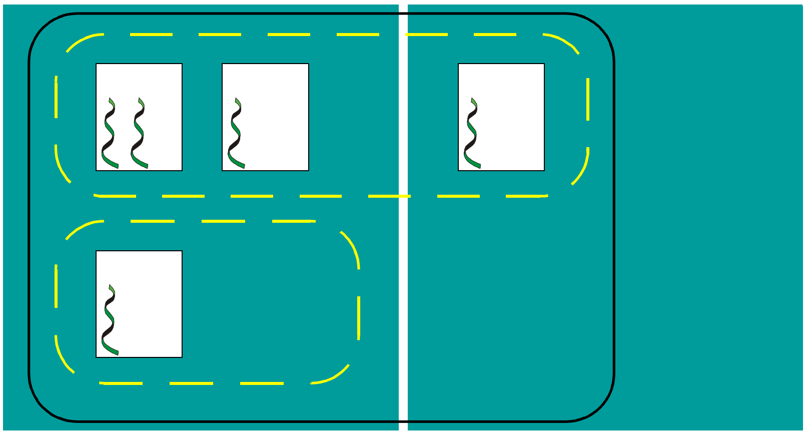

FIGURE 19: **Step 7**

8 A breakpoint is set on a line within a process executing in the first share group and the breakpoint is shared. The process executes until all three processes are at the breakpoint.

Control group: The group is unchanged.

Share group: The groups are unchanged.

Workers group: The group is unchanged.

Lockstep group: A lockstep group is created whose members are the four threads in the first share group. Because the thread in the second share group is still running, this control group only contains one lockstep group.

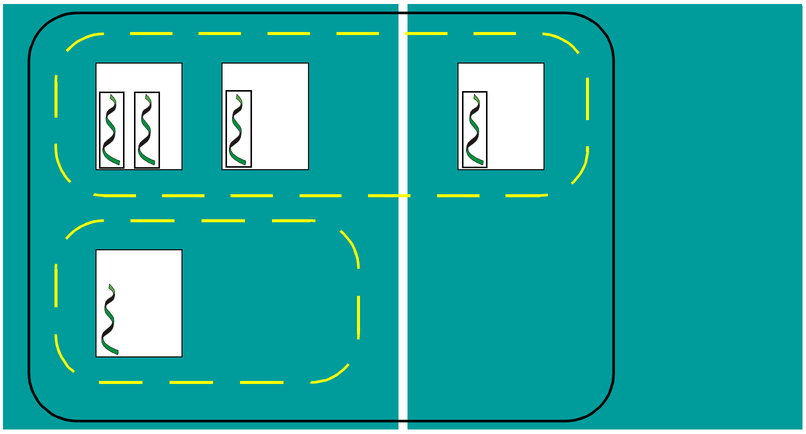

FIGURE 20: **Step 8**

9 You tell TotalView to step the lockstep group.

Control group: The group is unchanged.

Share group: The groups are unchanged.

Workers group: The group is unchanged.

Lockstep group: The group is unchanged.

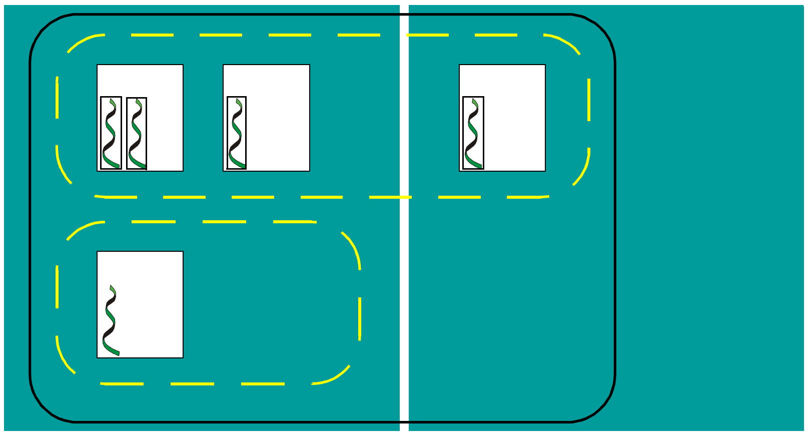

FIGURE 21: **Step 9**

Clearly, this example could keep on going until a much more complicated system of processes and threads was created. However, it should give you an idea of what is occurring.

More on Threads

It's time to say something pretty simple: the reason you are using a debugger is because your program isn't operating correctly and the way you think you're going to solve the problem (unless it is a &%$# operating system problem, which, of course, it usually is) is by stopping your program's threads, examining the values assigned to variables, and stepping your program so you can see what's happening as it executes.

Unfortunately, your multiprocess, multithreaded program and the computers upon which it is executing have lots of things executing that you want

TotalView to ignore. For example, you don't want to be examining manager and service threads created by the operating system, your programming environment, and your program.

Also, most of us are incapable of understanding exactly how a program is acting when perhaps thousands of processes are executing asynchronously. Fortunately, there are only a few problems that require full asynchronous behavior.

One of the first simplifications you can make is to change the number of processes. For example, suppose you have a buggy MPI program running on 100 processors. Your first step might be to have it execute in a 4 processor environment.

After you get the program running under TotalView's control, you will want to run the process being debugged to an action point, then stop it there so you can inspect the program's state at that place. In many cases, because your program has places where processes are forced to wait for an interaction with other processes, you can ignore what they are doing. That is, in most cases, you can let some processes keep on running because you don't care what they are doing (or perhaps you're sure that they are running correctly because you've already debugged them).

NOTE **TotalView lets you control as many groups, processes, or threads as you need to control. While each can be controlled individually, you will probably have problems remembering what you're doing if you're controlling large numbers of these things. The reason that TotalView creates and manages groups is so that you can focus on portions of your program.**

In most cases, you do not need to interact with everything that is executing. Instead, you want to focus on one process and the data that this process is manipulating. Where things get complicated is that the process being investigated is using data created by other processes, and these processes may have dependencies on other processes.

All this means that there is a rather typical pattern to the way you use TotalView to locate problems.

1. At some point, you'll want to make sure that you are interacting with the threads and processes you want to interact with. (You can do this using the CLI's **dgroupremove** command.)
2. You place an action point within a process or thread and begin investigating the problem. In many cases, you'll be setting an action point at a place where you hope the program is still executing correctly. At this time, you'll probably be wanting all processes and threads to operate synchronously. That is, you'll be creating process and thread barrier action points. You'll then start your program.
3. When execution stops at an action point, you'll look at the contents of your variables. At this time, you'll verify that your program state is actually correct.
4. If your program is correct, you'll begin stepping your program through its code. You'll probably want some sort of synchronous stepping or you'll want to set barriers so that everything isn't running freely.
5. Here's where things begin to get complicated. You've been focusing on one process or thread. If another process or thread is modifying the data and you become convinced that this is the problem, you'll want to go off to it and see what is going on.

The trick here, and it really isn't much of a trick, is to keep your focus narrow, investigating a limited number of behaviors. This is where debugging becomes an art. A multiprocess, multithreaded program can be doing a great number of things. Understanding where to look when problems occur is the "art".

If, for example, you are using the CLI to debug your program, you'll probably want to execute commands at the default focus. If you become convinced that the problem is in another process, you'll change to that process, also executing CLI commands in that process at their default focus.

In contrast, while you will often want to do something using another focus, what you will probably do is:

- Modify the focus so that it affects just the next command. For example, here's the command that steps thread seven in process three:

  ```
  dfocus t3.7 dstep
  ```

(In this example, the **dfocus** directive tells TotalView to limit the scope of what it does for the command that immediately follows and then, after the command completes, to restore the old focus.)

- Use the **dfocus** command to change focus temporarily, execute a few commands, then return to the original focus.

Setting Process and Thread Focus

When the CLI executes a command, TotalView has to decide which processes and threads it should act upon. Most commands have a default set of threads and processes and, in most cases, you won't want to make changes to it. Unfortunately, there are times when you'll need to change what the CLI is looking at. This section begins a rather intensive look at how you tell TotalView what processes and threads it should use as the target of a command.

Process/Thread (P/T) Sets

All CLI commands operate upon a set a processes and threads. This set is called a P/T (*Process/Thread*) *set*. A P/T set is a Tcl list containing one or more P/T identifiers. (The next section explains what a P/T identifier is.) Tcl lets you create lists in two ways:

- You can enter these identifiers within braces (**{ }**).
- You can use Tcl commands that create and manipulate lists.

These lists are then used as arguments to a CLI command. If you are entering one element, you usually do not have to use Tcl's list syntax.

For example, the following list contains specifiers for process 2, thread 1 and process 3, thread 2:

```
{ p2.1 p3.2 }
```

Unlike a serial debugger where each command clearly applies to the only executing thread, the CLI can control and monitor many threads and many different locations. The P/T *set* indicates the groups, process, and threads that are the target of the CLI command. No limitation exists on the number of groups, processes, and threads within a set.

In you do not explicitly specify a P/T set, the CLI defines a target set for you. This set is displayed as the (default) CLI prompt. (For information on this prompt, see "Command and Prompt Formats" on page 79.)

You can change the focus upon which a command acts using the e **dfocus** command. If the CLI executes a **dfocus** as a separate command, it changes the default P/T set. For example, if the default focus is process 1, the following command changes the default focus to process 2:

```
dfocus p2
```

If, however, **dfocus** is part of another command, it just changes the target for just the command that follows. After the command executes, the *old* default is restored.

The following example contrasts the two ways that you can use **dfocus** command. Assume that the current focus is process 1, thread 1. The following commands change the default focus to group 2 and then steps the threads in this group twice:

```
dfocus g2
dstep
dstep
```

Before the **dstep** command executes, it looks for the thread of interest in group 2. TotalView will then step all threads in the same group as the thread of interest.

In contrast, the following commands steps group 2, then steps process 1, thread 1:

```
dfocus g2 dstep
dstep
```

Some commands can only operate at the process level—that is, you cannot apply them to a single thread (or group of threads) in the process, but must apply them to all or to none.

Arenas

A P/T identifier often indicates a number of groups, processes, and threads. For example, assume that two threads executing the same program in process 2 are stopped at the same statement. This means that the two stopped threads form a lockstep group. If the default focus is process 2, stepping this process actually steps both of these threads.

The CLI uses the term *arena* to define the processes and threads that are the target of an action. In this case, the arena has two threads as you use the lockstep group as an argument within a **dfocus** command. Many CLI commands can act upon one or more arenas. For example, here is a command with two arenas:

```
dfocus p1 p2
```

The two arenas are process 1 and process 2.

Specifying Processes and Threads

A previous section said that a P/T set is a list. This ignored what the individual elements of the list are. A better definition is that a P/T set is a list of arena specifiers where an *arena* is the processes, threads, and groups that are affected by a CLI debugging command. Each *arena specifier* describes a single arena in which a command will act; the *list* is just a collection of arenas. Most commands iterate over the list, acting individually on an arena. Some output commands, however, may combine the arenas and act on them as a single target.

An arena specifier includes a *width specifier* and a *thread of interest*. ("Width specifiers" are discussed later in this section.)

Within the P/T set, the *thread of interest* specifies a target thread, while the width specifies how many threads surrounding the thread of interest are affected.

The thread of interest is specified as **p.t** where **p** is the TotalView process ID (PID) and **t** is the TotalView thread ID (TID).

The **p.t** combination identifies the process and thread of interest. The thread of interest is the primary thread that is affected by a command. For example, the **dstep** command always steps the thread of interest, but it may optionally run the rest of the threads in the process of interest and may step other processes in the group.

The CLI has two symbols with special meaning when specifying P/T sets:

>	The less-than symbol (<) character in place of the TID to indicate the *lowest number worker thread* in the process. If, however, the arena explicitly names a thread group, < means the lowest numbered member of the thread group. This symbol lets TotalView select the first user thread, which may not be thread 1; for example, the first and only user thread may be thread number 3 on Compaq systems.
.	A period indicates the current set. While this is seldom needed interactively, it can be useful in scripts.

Process and Thread Widths

You can enter P/T set in two ways. If you are not manipulating groups, the format is:

[*width_letter*][PID][.*thread_indicator*]

NOTE The next section extends this format to include groups.

For example, **p2.3** indicates process 2, thread 3. This width representation may look peculiar as it indicates that you do not have to specify anything. Because the CLI has an extensive set of defaults, it will try to fill in pieces that you omit. The only requirement here is that when you use more than one element, you use it in the order shown in this representation.

The *width_letter* indicates which processes and threads are part of this arena specifier. These letters are:

t *Thread width*

A command's target is the indicated thread.

p *Process width*

A command's target is the process containing the thread of interest.

g *Group width*

A command's target is the process containing the thread of interest contained within the group.

a *All processes*

A command's target is all threads in the group of interest that are in the process of interest.

d *Default width*

A command's target depends on the default for each command. This is also the width to which the default focus is set. For example, the **dstep** command defaults to process width (run the process while stepping one thread), and the **dwhere** command defaults to thread width (backtrace just one thread). Default width's are listed in "CLI Command Focus" on page 265.

These widths must be entered as lowercase letters.

The following figure illustrates the relationship of these specifiers:

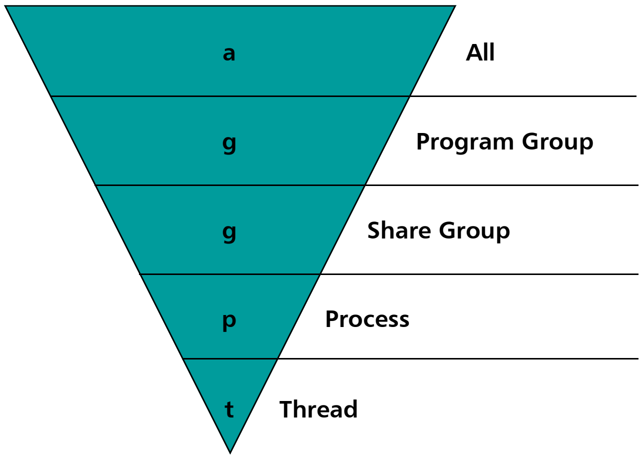

FIGURE 22: **Width Specifiers**

NOTE Notice that the "g" specifier indicates control and share groups.

You can visualize this relationship a triangle with its base on the top to indicate that the arena focusses on a greater number of entities as you move from thread level at the bottom to "all" level at the top.

CLI commands differ in what the target of their action can be:

- Some commands operate only (or primarily) on the thread of interest or process of interest.

- Other commands may extract the group from the arena and operate on that group.
- Some commands operate on all of an arena's processes and threads. An example is **dstatus**. (The **dstatus** command shows the status of processes and threads.) Command like **dstatus** use the width indicator to create a slice through the arena's processes and threads, and then operate on this slice.

As mentioned previously, the thread of interest specifies a particular target thread, while the width specifies how many threads surrounding the thread of interest are affected. For example, the **dstep** command always requires a thread of interest, but entering this command can:

- Step just the thread of interest during the step operation (single-thread single-step).
- Step all threads in the process containing the thread of interest (process-level single-step).
- Step all processes in the group that have threads at the same PC (program counter) as the thread of interest (group-level single-step).

This list doesn't include what happens to other threads related to the thread of interest. For more information, see "Bounded Stepping Commands" on page 57.

To save a P/T set definition for later use, assign the specifiers to a Tcl variable. For example:

```
set myset { g2.3 t3.1 }
dfocus $myset dgo
```

The thread of interest can also be modified by a *width* specifier. As the dfocus command returns its focus set, you can save this value for later use. For example:

```
set save_set dfocus
```

Examples

Here are some examples:

d1.<	Use the default set for each command, focusing on the first user thread in process 1. The < sets the TID to be the first user thread.
g2.3	Select process 2, thread 3 and set the width to group.

t1.7 Commands act only on thread 7 of process 1.

You can leave out parts of the P/T set if what you enter is unambiguous. A missing width or PID is filled in from the current focus. A missing TID is always assumed to be <. For more information, see "Incomplete Arena Specifiers" on page 53.

Setting Group Focus

When you start a multiprocess program, the CLI adds each process to a process group as the process starts. The debugger groups the processes depending on the type of system call (**fork()** or **execve()**) that created or changed the processes. There are two different types of process groups:

- **Control Group**

  Contains the parent process and all related processes. A control group includes children that were forked (processes that share the same source code as the parent) and children that were forked but which subsequently called **execve()**. That is, the children of the created processes that do not share the same source code as the parent.

  Assigning a new value to the **CGROUP()** variable for a process changes the program group for that process. In addition, the **dgroupadd** command lets you add members to a group.

- **Share Group**

  Contains the related processes that share the same source code.

  A share group contains all members of a control group that share the same executable image. (Note, however, that dynamically loaded libraries may vary between share groups member.)

  TotalView automatically places processes in share groups based on their program group and their executable image. You can't change a share group's members.

In addition, there are also two types of thread groups:

- **Workers Group**

  Contains all worker threads from all processes in the control group. By default, it contains all threads except the kernel-level manager threads that can be identified. You can use all group manipulation commands on workers group. However, you cannot delete them.

- **Lockstep Group**

  Contains every stopped thread in a share group that have the same PC. There is one lockstep group for every thread.

  The group ID's value for a lockstep group differs from the ID of other groups. Rather than an automatically allocated integer ID, the lockstep group ID has the form **pid.tid**, where **pid.tid** identifies the thread with which it's associated. For example, the lockstep group for thread 2 in process 1 is **1.2**.

Process groups can only contain processes. Thread groups can only contain threads, but threads in a thread group can be from more than one process. While the two group types can usually be used interchangeably, some commands can act differently depending on the kind of group is being manipulated.

In general, if you are debugging a multiprocess program, the control group and share group differ only when the program has children that are forked with a call to **execve()**.

Specifying Groups in P/T Sets

The arena specifier can also include a target group. If you do not include a group specifier, the default is the control group. The CLI only displays a target group in the focus string if you set it to something other than the default value.

NOTE Target group specifiers are most often used with the single step commands as they give these command more control over what is being stepped.

Here is how you add a groups to the way you specify arenas.

[*width_letter*][*group_indicator*][PID][.*thread_indicator*]

This format adds the *group_indicator*to the previously discussed syntax. There are actually several different ways that you can indicate a group.

- You can name one of TotalView's predefined sets. These sets are identified by letters. For example, the following command sets the focus to the workers group.

dfocus W

- You can identify a group by its number. For example, here is how you set the focus to group 3.

  dfocus 3/

  Notice the trailing slash. This slash lets the CLI know that you are specifying a group number instead of a PID. For example, the following names group 3 within process 3.

  dfocus 3/3

- As you can also name sets, you must surround these set names with slashes. For example, here is how you would name the **interesting** set of threads within process 2:

  dfocus p/interesting/2

The complete representation for the P/T set syntax may look peculiar because it indicates that you do not have to use a specifier, which is correct. If you do not use one, the CLI obtains the arena from the current P/T set focus. On the other hand, you can use any combination of these entities. The only rule is that the order for specifying these terms is as shown in the syntax representation.

As you will see, you can use either a *group_letter* and a *group_number* or you can use both of these options within one specifier. The *group_number* is a value that TotalView assigns to the group.

The *group_letter* can be:

C *Control group*

All processes in the control group.

D *Default program group*

The default group ID, indicating all processes in the control group. This differs from a **C** group specifier in that **D** clears the group letter from the CLI prompt. This is the default group ID.

S *Share group*

The set of processes in the control group that have the same executable as the arena's thread of interest.

W *Workers group*

The set of all worker threads in the control group.

L *Lockstep group*

A set containing all threads in the share group that have the same PC as the arena's thread of interest. If these threads are stepped as a group, they will proceed in lockstep.

You can only use uppercase letters for the group letter.

The slash character is optional if you are using a *group_letter*. However, you must use it as a separator when entering a numeric group ID and a **pid.tid** pair. For example, the following entry indicates workers group 3 in process 2:

```
pW3/2
```

The following table indicates what specifier combinations mean for the CLI's stepping commands:

TABLE 2: **Specifier Combinations**

Specifier	Meaning
aC	Specifies all process in all control groups.
aS	Specifies all threads in all share groups.
aW	Specifies all threads in all workers groups.
aL	Specifies all threads in all lockstep groups.
gC	Specifies all threads in the thread of interest's control group.
gS	Specifies all threads in the thread of interest's share group.
gW	Specifies all worker threads in the control group containing the thread of interest.
gL	Specifies all threads in the same share group within the process containing the thread of interest that have the same PC.
pC	Specifies all thread's in the control group of the process of interest. This is the same as **gC**.
pS	Specifies all threads in the process that participate in the same share group as the thread of interest.
pW	Specifies all worker threads in the process process of interest.
pL	Specifies all threads in the process of interest whose PC is the same as the thread of interest.

TABLE 2: Specifier Combinations

Specifier	Meaning
tC tS tW tL	These four combinations, while syntactically correct, are meaningless. The **t** specifier overrides the group specifier. So, for example, **tW** and **t** both name the current thread.

<u>* The list of specifier combinations needs to be validated!</u>***

NOTE On some systems, TotalView cannot distinguish manager threads from user threads, some manager threads may be chosen by mistake. This means that you may need to remove these manager threads from groups using the dgroupremove command.

Here are a few examples:

pW3	All worker threads in process 3.
pW3.<	All worker threads in process 3. Notice that this is the same as the previous focus specifier.
gW3	All worker threads in the control group containing process 3. Notice the difference between this and **pW3**, which restricts the focus to one of the processes in the control group.
gL3.2	All threads in the same share group within process 3 that are executing at the same PC as thread 2 in process 3.
/3	Specifies processes and threads in process 3. As the arena width, process of interest, and thread of interest are inherited from the existing P/T set, the exact meaning of this specifier depends on the previous context.
p4/1	All processes and threads in process 1 that are in group 4. Group 4 can either by a process or thread group.
g3.2/3	The 3.2 group ID is a synonym for the lockstep group for thread 3.2. This group includes all threads in process 3's share group that are executing at the same PC as thread 2.

NOTE Specifying thread width with an explicit group ID probably doesn't mean much.

In the following examples, the **dfocus** command creates a temporary P/T set (the second item in the list) upon which the CLI command (the last term) will operate. The **dstatus** command lists information about processes and threads.

dfocus g dstatus
: Displays the status of all threads in the control group.

dfocus gW dstatus
: Displays the status of all worker threads in the control group.

dfocus pW dstatus
: Displays the status of all worker threads in the current focus process. The width is process level and the target is the workers group.

dfocus p dstatus
: Displays the status of all worker threads in the current focus process. The width here, as in the previous example, is process and the (default) group is the control group; intersecting this width and the default group creates a focus that is also the same as the previous example.

The following example shows how the prompt changes as you change the focus. Notice how the prompt changes when using the **C** and the **D** group specifiers.

```
d1.<> f C
dC1.<
dC1.<> f D
d1.<
d1.<>
```

Setting Groups

This section presents a series of examples that set and create groups. Many of the examples use CLI commands that have not yet been introduced. You will probably need to refer to the command's definition in Chapter 5 before you can appreciate what is occurring.

Here are six methods by which you can indicate that thread 3 in process 2 is a worker thread.

dset WGROUP(2.3) $WGROUP(2)
: Assigns the group ID of the thread group of worker threads associated with process 2 to the **WGROUP** variable. (Assigning a non-zero value to **WGROUP** indicates that this is a worker group.)

dset WGROUP(2.3) 1
: A simpler way of doing the same thing as the previous example.

dfocus 2.3 dworker 1
: Adds the groups in the indicated focus to a workers group.

dgroupadd -g $WGROUP(2) 2.3
: Adds process 2, thread 3 to the worker group associated with process 2.

dfocus t2.3W dgroupadd
: A simpler way of doing the same thing as the previous example.

```
dset GROUP($WGROUP(2)) \
        [ concat $GROUP($WGROUP(2)) 2.3 ]
```
An extreme example of using related elements to mark a thread as being a worker thread.

dgroupadd -g AGroup -new 2.3
: Creates a group named **AGroup** and assigns process 2, thread 3 to it.

```
dset CGROUP(2) $CGROUP(1)
dgroupadd -g $CGROUP(1) 2
dfocus 1 dgroupadd 2
```
These three commands insert process 2 into the same control group as process 1.

Here are some additional examples.

dfocus g1 dgroupadd -new thread
: Creates a new thread group that contains all the threads in all the processes in the control group associated with process 1.

```
set mygroup [dgroupadd -new thread $GROUP($SGROUP(2))]
dgroupremove -g $mygroup 2.3
dfocus g$mygroup/2 dgo
```

Defines a new group containing all the worker threads in process 2's share group except for thread 2.3 and then continue that set of threads,. The first command creates a new group containing all the threads from the share group; the second removes thread 2.3; the third runs the remaining threads.

An Extended Example

When specifying an arena, the CLI has two places for specifying groups, each with a different purpose. Using **g** is required when you need to force the group while the current default focus indicates something different, For example, **gL** forces group width while **L**uses the width of the current focus.

The following example will clarify this difference. The first step is to set a breakpoint in a multithreaded OMP code and run it to a breakpoint:

```
d1.<> dbreak 35
Loaded OpenMP support library libguidedb_3_8.so :
                    KAP/Pro Toolset 3.8
1
d1.<> dcont
Thread 1.1 has appeared
Created process 1/37258, named "tx_omp_guide_llnl1"
Thread 1.1 has exited
Thread 1.1 has appeared
Thread 1.2 has appeared
Thread 1.3 has appeared
Thread 1.1 hit breakpoint 1 at line 35 in ".breakpoint_here"
```

The default focus is `d1.<`, which means that the CLI is at its default width, The process of interest (POI) is 1, and the thread of interest (TOI) is the lowest numbered non-manager thread. Because the default width for the **dstatus** (**st**) command is process, entering **st** tells the CLI to display the status of each of the processes. Notice that typing **dfocus p st** produces the same output:

```
d1.<> st
1:       37258Breakpoint[tx_omp_guide_llnl1]
```

```
        1.1:  37258.1BreakpointPC=0x1000acd0,
                            [./tx_omp_llnl1.f#35]
        1.2:  37258.2StoppedPC=0xffffffffffffffff
        1.3:  37258.3StoppedPC=0xd042c944
    d1.<> dfocus p st
    1:          37258Breakpoint[tx_omp_guide_llnl1]
        1.1:  37258.1BreakpointPC=0x1000acd0,
                            [./tx_omp_llnl1.f#35]
        1.2:  37258.2StoppedPC=0xffffffffffffffff
        1.3:  37258.3StoppedPC=0xd042c944
```

Here's what is displayed when you ask for the status of the lockstep group. The rest of this example will use the **f** abbreviation for **dfocus**.

```
    d1.<> f L st
    1:          37258Breakpoint[tx_omp_guide_llnl1]
        1.1:  37258.1BreakpointPC=0x1000acd0,
                            [./tx_omp_llnl1.f#35]
```

This command tells the CLI to get the status of the threads in thread 1.1's (the TOI) lockstep group. The **f L** command modifier narrows the set so that the display only includes the threads in the process that are at the same PC as the TOI.

NOTE By default, the dstatus command displays information at "process" width. This means that you do not need to type "f pL st".

The next command runs thread 1.3 to the same line as thread 1.1. This immediately followed by a command that displays the status of all the threads in the process:

```
    d1.<> f t1.3 duntil 35
        35@>       write(*,*)"i= ",i,
                 "thread= ",omp_get_thread_num()
    d1.<> f p st
    1:          37258Breakpoint[tx_omp_guide_llnl1]
        1.1:  37258.1BreakpointPC=0x1000acd0,
                            [./tx_omp_llnl1.f#35]
        1.2:  37258.2StoppedPC=0xffffffffffffffff
        1.3:  37258.3BreakpointPC=0x1000acd0,
                            [./tx_omp_llnl1.f#35]
```

As expected, the CLI has added a thread to the lockstep group:

```
d1.<> f L st
1:          37258Breakpoint[tx_omp_guide_llnl1]
    1.1:  37258.1BreakpointPC=0x1000acd0,
                      [./tx_omp_llnl1.f#35]
    1.3:  37258.3BreakpointPC=0x1000acd0,
                      [./tx_omp_llnl1.f#35]
```

The next set of commands first narrows the width of the default focus to thread width—notice that the prompt changes—then displays the contents of the lockstep group.

```
d1.<> f t
t1.<> f L st
1:          37258Breakpoint[tx_omp_guide_llnl1]
    1.1:  37258.1BreakpointPC=0x1000acd0,
                      [./tx_omp_llnl1.f#35]
```

This is the hard step. While the lockstep group has two threads, the current focus has only thread 1, and, coincidentally, that thread is part of the lock-step group. Consequently, the lockstep group *in the current focus* is just the one thread.

If you ask for a wider width (**p** or **g**) with **L**, the CLI displays more threads from the lockstep group.

```
t1.<> f pL st
1:          37258Breakpoint[tx_omp_guide_llnl1]
    1.1:  37258.1BreakpointPC=0x1000acd0,
                      [./tx_omp_llnl1.f#35]
    1.3:  37258.3BreakpointPC=0x1000acd0,
                      [./tx_omp_llnl1.f#35]
t1.<> f gL st
1:          37258Breakpoint[tx_omp_guide_llnl1]
    1.1:  37258.1BreakpointPC=0x1000acd0,
                      [./tx_omp_llnl1.f#35]
    1.3:  37258.3BreakpointPC=0x1000acd0,
                      [./tx_omp_llnl1.f#35]
t1.<>
```

Because this example only contains one process, the **pL** and **gL** modifiers produce the same result when used with dstatus. If, however, the program had additional processes in the group, you could only see them using a **gL** modifier.

In this example, the focus indicated by the prompt—this focus is called the outer focus—controls the display. Notice what happens when dfocus commands are strung together:

```
t1.<> f d
d1.<
d1.<> f tL st
1:          37258Breakpoint[tx_omp_guide_llnl1]
     1.1: 37258.1BreakpointPC=0x1000acd0,
                       [./tx_omp_llnl1.f#35]
d1.<> f tL f p st
1:          37258Breakpoint[tx_omp_guide_llnl1]
     1.1: 37258.1BreakpointPC=0x1000acd0,
                       [./tx_omp_llnl1.f#35]
     1.3: 37258.3BreakpointPC=0x1000acd0,
                       [./tx_omp_llnl1.f#35]
d1.<> f tL f p f D st
1:          37258Breakpoint[tx_omp_guide_llnl1]
     1.1: 37258.1BreakpointPC=0x1000acd0,
                       [./tx_omp_llnl1.f#35]
     1.2: 37258.2StoppedPC=0xffffffffffffffff
     1.3: 37258.3BreakpointPC=0x1000acd0,
                       [./tx_omp_llnl1.f#35]
d1.<> f tL f p f D f L st
1:          37258Breakpoint[tx_omp_guide_llnl1]
     1.1: 37258.1BreakpointPC=0x1000acd0,
                       [./tx_omp_llnl1.f#35]
     1.3: 37258.3BreakpointPC=0x1000acd0,
                       [./tx_omp_llnl1.f#35]
d1.<>
```

String multiple focuses together may not produce the most readable result, this example illustrates how one **dfocus** command can modify what another sees and will act upon. The ultimate result is an arena upon which a command will act. In these examples, the **dfcous** command is telling the **dstatus** command what it should be displaying.

Incomplete Arena Specifiers

In general, you do not need to completely specify an arena. Missing components are assigned default values or are filled in from the current focus. The only requirement is that the meaning of each part of the specifier cannot be ambiguous. Here is how the CLI fills in missing pieces:

- If you do not use a width, the CLI uses the width from the current focus.
- If you do not use a PID, the CLI uses the PID from the current focus.
- If you set the focus to a list, there is no longer a default focus. This means that you must explicitly name a width and a PID. You can, however, omit the TID. (If you omit the TID, the CLI defaults <.)
- You can type a PID without typing a TID. If you omit the TID, the CLI uses its default of "<" where "<" indicates the lowest numbered worker thread in the process. If, however, the arena explicitly names a thread group, < means the lowest numbered member of the thread group.

  In most cases, the CLI does not use the TID from the current focus, since the TID is a process-relative value.
- A dot typed before or after the number lets the CLI know if you are specifying a process or a thread. For example, "**1.**" is clearly a PID, while "**.7**" is clearly a TID.

  If you type a number without typing a period, the CLI interprets the number as being a PID.
- If the width is **t**, you can omit the dot. For instance, **t7** refers to thread 7.
- If you enter a width and do not specify a PID or TID, the CLI uses the PID and TID from the current focus.

  You can use a bare alphabetic group specifier. The CLI obtains the rest of the arena specifier from the default focus.
- You can use a group ID or tag followed by a "/". The CLI obtains the rest of the arena specifier from the default focus.

Lists With Inconsistent Widths

The CLI lets you create lists containing more than one width specifier. While this can be very useful, it can be confusing. Consider the following:

```
{ p2 t7 g3.4 }
```

This list being defined is quite explicit: all of process 2, thread 7, and all processes in the same group as process 3, thread 4. However, how should the CLI use this set of processes, groups, and threads?

In most cases, the CLLI does what you would expect it to do: a command iterates over the list and act on each arena. If the CLI cannot interpret an inconsistent focus, it prints an error message.

There are commands that act differently. These commands use each arena's width to determine the number of threads on which it will act. This is exactly what the **dgo** command will do. In contrast, the **dwhere** command creates a call graph for process-level arenas, and the **dstep** command runs all threads in the arena while stepping the thread of interest. It may wait for threads in multiple processes for group-level arenas.

Stepping

The action that the CLI will perform when stepping assembler instructions or source statements depends if you have specified thread, process, or group width.

NOTE If you do not explicitly name a group, the CLI steps the control group.

Here is what happens at each width:

- **Thread**

  TotalView steps the thread of interest. Stepping a thread is not the same as stepping a thread's process because a process can have more than one thread.

  NOTE **Thread stepping is not implemented on Sun platforms. On SGI platforms, thread stepping is not available with pthread programs. If, however, your program's parallelism is based on SGI's sprocs, thread stepping will be available.**

  Thread width (**t**) tells the CLI that it should just run that thread. In contrast, process width (**p**) tells the CLI that it should run all threads in the process that are allowed to run while the thread of interest is stepped.

  TotalView also allows all manager threads to run.

- **Process** (default)

  TotalView runs all threads in the process, and execution continues until the thread of interest arrives at its goal location, which can be the next statement, the next instruction, and so forth. Only when the thread of interest reaches the goal are the other threads in the process stopped.

- **Group**

  The CLI examines the group and identifies each process having a thread stopped at the same location as the thread of interest (a "matching" process). The CLI waits until the thread of interest and one thread from each matching process arrive at the action point.

  If you use a process group specifier (C, D, S, W, or L), the CLI examines the group to identify those processes having a thread stopped at the same location as the thread of interest (a "matching" process). After selecting one matching thread from each matching process, it runs all processes in the group, and waits until the thread of interest and each selected thread arrive at the goal location.

  If you use a thread group specifier, TotalView runs all threads in the group to the same goal as the thread of interest. (If a thread that is not in this group arrives at the goal, TotalView will also stop it.) The group of interest specifies the set of threads that the CLI will wait for—the command does not complete until all threads in the group of interest are at the goal.

  When you step a control group, only the process of interest runs. In contrast, stepping a thread group runs all processes in the control group.

  NOTE **The workers group includes all workers in the control group, not just the worker threads from a single process.**

  Regardless of which threads are in the group of interest, the CLI will only wait for threads that are in the same share group as the thread of interest.

If a thread hits an action point other than the goal breakpoint during a step operation, the step ends.

The **duntil** command differs from other step commands when it's applied to a group. (The **duntil** command tells TotalView to execute program statements *until* a selected statement is reached.) This command runs the entire group and the CLI waits until all processes in the group have at least one thread arrives at the goal breakpoint. This lets you *sync up* all the processes in a group in preparation for group-stepping them. If you use **duntil** with a thread group, TotalView runs the process (for **p** width) or the control group (for **g** width) and waits until all the running threads in the group of interest arrive at the goal.

In all cases, if the focus process does not exist before a command executes, TotalView creates the process and then executes the command.

Bounded Stepping Commands

The bounded stepping commands are **dstep**, **dstepi**, **dnext**, **dnexti**, **dout**, and **duntil**. (They are *bounded* because they run to a fixed goal.) They all run one or more threads to a goal. They all execute synchronously—the CLI is blocked while each runs.

Stepping When No Target Group is Specified

Here are some of the operations that can occur when using bounded stepping commands:

******* Please validate examples.***

- **Run a single thread to a goal**

  While the thread runs, no other thread runs (except kernel manager threads). Optionally, the CLI may run all non-worker threads.

  Example: **dfocus t dstep**
- **Run a single thread to a goal while the process runs**

  A single thread runs into or through a critical region.

  Example: **dfocus p dstep**
- **Run one thread in each process in a share group to a goal**

  While one thread in each process in a share group runs to a goal, the rest of the threads run freely.

  Example: **dfocus gS dstep**
- **Run all worker threads in the process to a goal while non-worker threads run**

  Runs symmetric worker threads through a parallel region in lockstep.

  Example: **dfocus gW dstep**
- **Run all workers in the share group to a goal**

  All processes in the share group participate. The non-worker threads run.

  Example: **dfocus gS dstep**

- **Run all threads that are at the same PC as the thread of interest to a goal**

  The CLI selects threads from one process or from the entire share group. This is very much like the previous two bullets, but the CLI uses the set of threads which are in lockstep with the thread of interest rather than using the workers group.

  Example: dfocus L dstep

Here is how the bounded step commands behave when operating at process and group widths:

- **Process Group Operation**: Examine the thread of interest to determine the goal, and then run the entire control group until the focus thread and one thread from each process in the focus group arrives at the goal.
- **Group-Width Thread Group Operation**: Run the entire control group. The CLI commands wait for the threads in the focus group to reach the goal. After all focus group threads arrive, TotalView stops the control group again (and the original run state is restored).
- **Process-Width Thread Group Operation**: Run the entire process. CLI commands wait for just the threads in the focus group to reach the goal.

The **duntil** command behaves differently. For more information, see "duntil" on page 216.

Stepping With an Explicit Group and Arena Width Tag

The arena width specifiers (**a**, **g**, **p**, and **t**) name the set of processes and threads that will run. The group specifiers (**C**, **S**, **W**, **L**, or a numeric group ID) name the set of threads that will be run to the goal. (As usual, the **duntil** command behaves differently.)

When you use a control group or another process group, the thread of interest is run to a goal along with one thread from each participating processes. Other threads in the control group run freely.

NOTE In general, trying to run all the threads in a process to a goal doesn't work

When you use a thread group (**W**, **L**, or an explicit thread group ID), TotalView determines which threads are run to the goal by intersecting the specified group and the arena width. All other threads in the control group run freely during the operation.

TotalView defines the set of threads that are run to the goal from the thread of interest's share group.

The default focus uses the full control group (**C**). This means that one thread runs to a goal and the others run freely.

- **Group-Width Arena with a Control Group (C)**: The CLI searches the share group for threads that match the PC of the thread of interest. TotalView runs one matching thread from each share group member to the goal along with the thread of interest. All other threads in the control group run freely.
- **Process-Width Arena with a Workers Group (W)**: TotalView runs all worker threads in the process to the same goal. Non-worker threads in the process also run. This lets you step the workers together through a parallel region in Single Instruction, Multiple Data (SIMD)-style parallel programs. Uniform system-style programs—that is, programs having a varying set of threads that operate on similar but not necessarily identical tasks—are not handled as easily by this mode.
- **Group-Width Arena with a Workers Group (W)**: TotalView runs all worker threads in the share group to the goal; all other threads in the program group also run. This lets you step all workers across a multiprocess multithreaded program through a parallel region together.
- **Process-Width Arena with the Lockstep Group (L or GID = pid.tid)**: TotalView runs all threads having the same PC as the thread of interest to the goal; other threads in the process run freely. The threads are *lined up* at the start of a parallel region. This allows you to step them through the region in lockstep without having to worry about which are in the workers group.
- **Group-Width Arena with the Lockstep Group**: TotalView runs all threads having the same PC in the share group to the goal; other threads in the control group run freely. This lets you run all threads executing the same code in a multiprocess, multithread program through a parallel region together without having to worry about which ones are in the workers group.

Some Examples

In the following examples, the default focus is set to **d1.<**.

dstep	Steps the thread of interest while running all other threads in the process.
dfocus W dnext	Runs the thread of interest and all other worker threads in the process to the next statement. Other threads in the process run freely.
dfocus W duntil 37	Runs all worker threads in the process to line 37.
dfocus L dnext	Runs the thread of interest and all other stopped threads at the same PC to the next statement. Other threads in the process run freely. Threads that encounter a temporary breakpoint in the course of hopping to the next statement usually join the lockstep group.
dfocus gW duntil 37	Runs all worker threads in the share group to line 37. Other threads in the program group run freely.
UNW 37	Performs the same action as the previous command: runs all worker threads in the share group to line 37. This example uses the predefined **UNW** alias instead of the individual commands. That is, **UNW** is an alias for **dfocus gW duntil**.
SL	Finds all threads in the share group that are at the same PC as the thread of interest and step them all one statement. This command is the built-in alias for **dfocus gL dstep**.
sl	Finds all threads in the current process that are at the same PC as the thread of interest, and step them all one statement. This command is the built-in alias for **dfocus L dstep**.

"Piling Up" versus "Running Through"

When you enter a step command with its focus set to the lockstep group, the CLI runs all threads at the same PC as the thread of interest to the same goal. In contrast, when you enter a step command with its focus set to a non-lockstep thread group, the CLI runs all threads in the *indicated group* to the same goal as the thread of interest. The set of threads being run to the goal is called *the active set*. In all cases, the goal is selected based on the PC of the thread of interest. The step command then waits for all threads in the active set to get to that goal.

While the active set is running to the goal, all other threads in the process (or group) run freely. This includes all non-worker threads, and it may include some worker threads if the active set does not include all workers.

If a thread that is not in the active set reaches the goal breakpoint, the CLI continues the process again until all active threads reach the goal. Before continuing the process, the CLI can either step the thread over the goal breakpoint or it can hold it.

If a single thread is being run into a critical region, threads that are not in the active set run freely. Otherwise, the thread of interest may not be able to make any progress.

If a lockstep group is running, threads that are not in the active set pile up at the goal when the CLI steps the lockstep group. The CLI does not step them over the goal breakpoint if you had specified a thread group.

"Piling up" can only occur for system upon which the CLI can control asynchronous threads.

P/T Set Expressions

At times, you don't want all of one kind of group or process to be displayed. The CLI lets you use the following three operators to manage your P/T sets:

|	Creates a union; that is, all members of the sets.
-	Creates a difference; that is, all members of the first set that are not also members of a second set.
&	Creates an intersection; that is, all members of the first set that are also members of the second set

For example, here is how you would create a union of two P/T sets:

```
dfocus p3 | L2
```

As the definitions imply, these operators only work on two sets. However, you can apply these operations repeatedly. For example:

```
dfocus p2 | p3 & L2
```

This statement creates a union between **p2** and **p3**, then creates an intersection between this result and **L2**.

The CLI associates sets from left to right. You can change this order using parentheses. For example:

```
dfocus p2 | (p3 & pL2)
```

Typically, these three operators are used with the following P/T set functions:

breakpoint	Returns a list of all threads that are stopped at a breakpoint.
error	Returns a list of all threads stopped due to an error.
existent	Returns a list of all threads.
held	Returns a list of all threads that are held.
nonexistent	?????

******* What does nonexistent mean?***

running	Returns a list of all running threads.
stopped	Returns a list of all stopped threads.
unheld	Returns a list of all threads that are not held.
watchpoint	Returns a list of all threads that are stopped at a watchpoint.

The following examples should clarify how these operators and functions are used:

f {breakpoint(a) | watchpoint(a)} dstatus
: Show all threads that stopped at breakpoints and watchpoints.

f { stopped(a) - breakpoint(a) } dstatus
: Show all stopped threads that are not stopped at breakpoints

f { g.3 - p6 } duntil 577

Run threads three along with all other processes in the group to line 577. However, do not run anything in process 6.

f { ($PTSET) & p123)

Use just process 123 within the current P/T set.